

Incremental Entity Resolution from Linked Documents

Pankaj Malhotra, Puneet Agarwal, Gautam Shroff

TCS Research

Tata Consultancy Services Ltd., Sector 63

Noida, Uttar Pradesh, India

{malhotra.pankaj, puneet.a, gautam.shroff}@tcs.com

February 19, 2014

Abstract

In many government applications we often find that information about entities, such as persons, are available in disparate data sources such as passports, driving licences, bank accounts, and income tax records. Similar scenarios are commonplace in large enterprises having multiple customer, supplier, or partner databases. Each data source maintains different aspects of an entity, and resolving entities based on these attributes is a well-studied problem. However, in many cases documents in one source reference those in others; e.g., a person may provide his driving-licence number while applying for a passport, or vice-versa. These links define relationships between documents of the same entity (as opposed to inter-entity relationships, which are also often used for resolution). In this paper we describe an algorithm to cluster documents that are highly likely to belong to the same entity by exploiting inter-document references in addition to attribute similarity. Our technique uses a combination of iterative graph-traversal, locality-sensitive hashing, iterative match-merge, and graph-clustering to discover unique entities based on a document corpus. A unique feature of our technique is that new sets of documents can be added incrementally while having to re-resolve only a small subset of a previously resolved entity-document collection. We present performance and quality results on two data-sets: a real-world database of companies and a large synthetically generated ‘population’ database. We also demonstrate benefit of using inter-document references for clustering in the form of enhanced recall of documents for resolution.

1 Introduction

We address a commonly occurring situation where information about entities, such as persons, are maintained in disparate data sources each describing different aspects of an entity. Our goal is not only to identify the documents belonging to an entity, but also to merge these documents in order to create one document per entity. For example, information about residents of a country is available in multiple disparate databases like passports, driving licenses, bank accounts, phone-connection records, and income-tax records.

We focus on the fact that often there are cases when one type of document of an entity may contain a reference to another type of document. For example, a passport document may refer to the driving license id of the person as an identity proof. Such references can be leveraged to identify different documents belonging to the same person, as we describe in our approach for ‘entity resolution from linked documents’.

The problem of entity resolution has been addressed in many ways. Many approaches to entity resolution problem view it as a problem of data de-duplication [1], duplicate detection [2], or data cleaning [3]. In these cases the goal is to detect and remove duplicate records for the same real-world entity, merge-purge [4], or link records [5]. Collectively resolving entities based on relationships between *different* entities has been addressed in [6]. In contrast, we focus on references between documents that belong to the *same* entity, and show how this additional information can be used to enhance the accuracy and performance of entity resolution.

Explicitly comparing all pairs of documents is of quadratic complexity and becomes infeasible for even moderately sized collections. To achieve scalability ‘blocking’ techniques are generally used [7] wherein blocks of documents are obtained such that each block contains a relatively small number of potentially matching documents. Thereafter only documents within each block are exhaustively compared with each other (so the number of comparisons done is greatly reduced) to find matching documents (i.e., documents belonging to same entity). Hence, blocking based entity resolution is usually considerably more efficient than an all-pairs comparison. **We show how blocking efficacy can be enhanced by exploiting inter-document references.**

In many real-life situations new sets of documents keep arriving incrementally over time. These documents may pertain to new entities, i.e., the entities that do not exist in the database of resolved entities at the time these documents arrive, or may pertain to

already existing entities. New documents could even potentially alter previously taken decisions about which sets of documents belong to different entities. Thus the database of resolved entities needs to be appropriately updated with this new information. **We show how to update the resolved set of document-entity pairs efficiently, without having to re-resolve the entire collection.**

The rest of the paper is organized as follows: We begin by formally describing the linked documents scenario in Section 2. In Section 3 we explain the key motivations for our solution and introduce our approach to resolving entities from linked documents, which we call ERLD. In Section 4 we describe the ERLD algorithm in detail and its incremental variant in Section 5.

In Section 6 we present performance results on two data-sets: a real-world database of companies, and a large synthetically generated ‘population’ database. We show how leveraging the inter-document references enhances the quality of the solution by improving recall. We also demonstrate the scalability of ERLD as compared to an pairwise iterative match merge approach [8]. Finally we show that resolving a new set of documents against a pre-resolved set of entities can be done efficiently using Incremental ERLD. We conclude in Section 8 after a brief description of related work in Section 7.

2 Linked Documents

We assume that information about real-world entities is available from disparate data sources in the form of documents linked by references, formally defined as follows:

Document: A *document* consists of *attributes*, where each *attribute* has a unique name and zero or more values. Each document belongs to a unique real-world *entity*. For example, a passport has attributes *name*, *father’s name*, *address*, and *date-of-birth*; additionally, one’s *driving licence number* may be given as identity proof.

Match: Two documents are said to Match if they return `true` under some *match function*.

Match function: A Boolean function defined over two documents that returns `true` when two *documents* are deemed as belonging to the same entity, and `false` otherwise. Match functions can be implemented in different ways. For example, they can be based on rules defined over the attribute values of the two documents being compared: `true` if name matches AND address matches AND date-of-birth matches, `false` otherwise. We use such a rule-based Match function in our experiments on one of the data-sets (refer Section 6.1). Match functions may also be based on classifiers derived via machine learning: Features for such a classifier can be the scores obtained by comparing the attribute values of the corresponding attributes in the two documents. For example, the Jaccard similarity score can be used to compare name attributes from two documents. Scores from different such similarity measures become features to classify the pairs of documents as `true` (matching) or `false` (non-matching) categories, via machine learning. We use a Support Vector Machines (SVM) classifier [9] in another of our case-studies (refer Section 6.2).

Merge: A procedure that takes two or more documents as input and produces a new document that has attributes and their values coming from all the input documents. Note that the resulting merged document may have multiple values for some or even all its attributes.

Entity: We use the term entity in two senses: (i) real-world entities such as persons and (ii) clusters of documents that are likely to belong to the same real-world entity. Thus, an entity of the type person has (potentially multi-valued) attributes such as name, father’s name, address, date-of-birth, etc. The values of these attributes are gathered from disparate documents associated with the same entity based on a Match function.

Document attributes differ depending what it means for different values of a particular attribute to be considered ‘matching’; in this context we define *types* of attributes as follows:

Soft Attribute: An attribute for which two values may be considered as matching under some matching criterion, even if they are not textually same/equal. For example, different variations of a person’s *name* (as shown in Figure 1) can be considered to be matching even if they are not textually identical.

Hard Attribute: An attribute for which two values are considered to be matching only if they are textually identical: For example, *Phone number* and *email-id* are hard attributes in Figure 1.

Unique Attribute: An attribute that has a unique value for each real-world entity. Clearly a unique attribute must also be hard, i.e., two values are considered to be matching only if they are textually identical. For a real-world entity possessing multiple documents, the value of any particular unique attribute must be the same in all of them (apart from cases of deliberate obfuscation or data errors). Also, two entities cannot have the same value for a unique attribute. For example, an entity can have only one *passport number*, and no two entities can have same *passport number*. Unique hard attributes are potential primary keys of a document. The idea of unique attributes has been defined in a similar way in [10], where an attribute has a hard uniqueness constraint if it can take a unique value or no value for each real-world entity.

Note that an entity *can* have multiple values for a soft or hard attribute, but not for a unique attribute. For example, a person can have multiple names and phone numbers but not multiple passport numbers. Further, different entities can have same value for a soft attribute or a hard attribute but not for a unique attribute. For example, two people can have same name (soft attribute) and can share same phone number (hard attribute) but not same passport number (unique attribute). On the other hand, violations of the uniqueness property for values of a unique attribute can be an indication of deliberate obfuscation, e.g., a person wanting to hide under multiple identities, or of data errors in a poor quality database. The implications of uniqueness and how it can be exploited may differ, e.g., for entity resolution or alternatively for detecting fraud or other suspicious behaviour.

Entity ID	ID	Document ID	Name (SA*)	Email ID (HA*)	Phone Number (HA*)	Date of Birth (SA*)	Address (SA*)	Proofid (RA*)	Document Details (RA*)
e ₁	d ₁	PAN11	J B Smith	NULL	NULL	NULL	ABC	VOT21	
e ₁	d ₂	VOT21	John Smith	NULL	NULL	AUG/17/1977	ABC		
e ₁	d ₃	DL32	John B Smith	a@x.com	510	17-08-1977	ABD	VOT21	
e ₁	d ₄	BAN64	John Blake Smith	a@x.com	510	08/17/1977	ABC		
e ₂	d ₅	PAN57	W L Schneider	b@x.com	951	24-11-1962	XYZ		
e ₂	d ₆	BAN26	Winifred Lela Schneider	b@y.com	NULL	NULL	XYZ		Driving Licence ID :DL77
e ₂	d ₇	DL77	Winifred Schneider	b@y.com	951	24-11-1962	PQR		
e ₃	d ₈	VOT89	Jacobson Ruiz	c@g.com	NULL	24-6-1969	MNO		
e ₃	d ₉	BAN91	J E Ruiz	c@g.com	888	24/06/1969	RST		
e ₃	d ₁₀	PAN68	Jacobson Ruiz	c@g.com	838	24-6-1969	MNO		
e ₄	d ₁₁	DL11	Carla Ruiz	NULL	848	NULL	ABZ		Father's Account Number: BAN91

Figure 1: Sample Linked Documents. Here, SA*:Soft Attribute, HA*: Hard Attribute, RA*: Referential Attribute

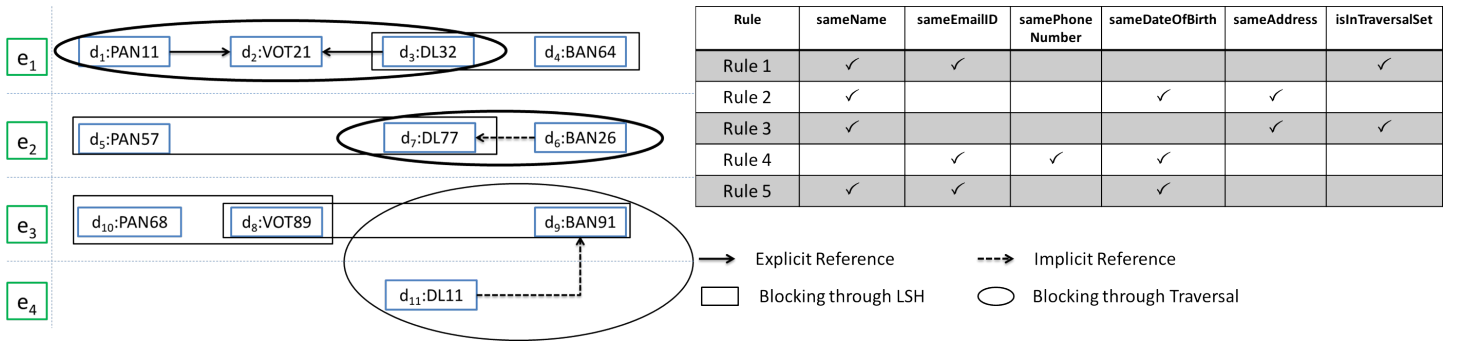


Figure 2: A pictorial view of Sample Linked Documents of Figure 1 with Rules for Matching

Referential Attribute: An attribute of a document that contains the value of a hard or unique attribute of *another* document. For example, an attribute named *Driving Licence ID* in a passport document can be considered as a referential attribute. A hard attribute can also be a referential attribute. For example, the *phone number* (a hard-attribute) of a person in a bank-account document may be the primary-key of the phone-connection document for that person. The *Proof ID* and *Document Details* attributes in Figure 1 contain inter-document references, and can be considered to be referential attributes.

We classify *referential attributes* into two types: (1) Explicit Referential Attributes, (2) Implicit Referential Attributes: The value of an explicit referential attribute exactly the value of a hard or unique attribute. On the other hand, for an implicit referential attribute, a *part* of its value *contains* a hard or unique attribute of another document. It is unknown which part of the value of an implicit referential attribute is the actual reference. For example, if a passport document has an attribute named *Driving Licence ID* with value equal to “DL123”, then the passport document makes an explicit reference to the *driving licence* document with primary key “DL123”, whereas if the value of the ‘description’ field in the passport document is “Applicant’s DL# DL123”, then the reference to the driving license document is implicit. For the example in Figure 1, *Proof ID* is an explicit referential attribute whereas *Document Details* can be an implicit referential attribute, when, for example it *contains* the value “BAN91” as part of its textual content.

A document referring to another document is likely to indicate one of the two things: (1) documents belong to the same entity, (2) documents belong to different entities which are related to each other. For example, a description field in a bank-account document containing a mobile number may imply that the mobile number referred is of the same person or that of a relative (such as a parent) of the person. If this mobile number is also present in another document, then the two documents potentially belong to the same person or two persons who are related to each other. However in this paper we do not consider or exploit *explicit* references between documents belonging to different entities. Further *implicit* references to other entities are treated as noise for our purposes.

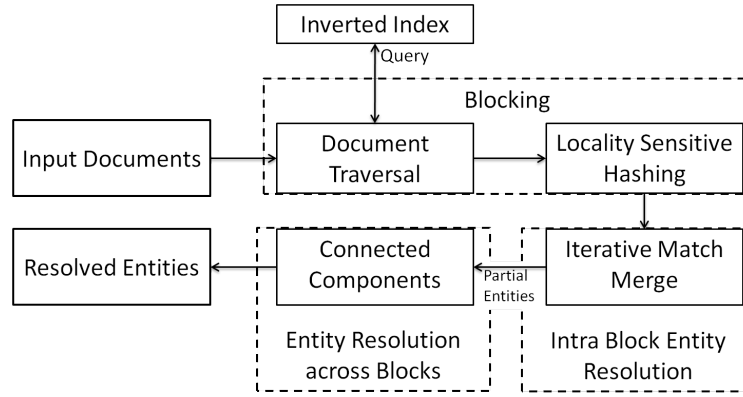


Figure 3: Entity Resolution from Linked Documents (ERLD)

3 Entity Resolution from Linked Documents

Entity Resolution (ER): Given a collection of *documents* $D = \{d_1, d_2, \dots, d_N\}$, where each *document* d_i belongs to a unique *entity*, determine a collection of entities $E = \{e_1, e_2, \dots, e_K\}$ where each e_i is obtained by applying the *Merge* function on a subset of document(s) coming from D that *Match* under a given *Match* function. For example, different of documents belonging to the same person, such as Voter-Card, Passport, and Driving-Licence, will get merged to form one entity.

3.1 Overview of ERLD Algorithm

The goal of our **Entity Resolution from Linked Documents** algorithm (ERLD) is to group documents into entities so that all the documents in an entity correspond to the same real-world entity. When the number of documents that need to be resolved is large, comparing every document with every other document to group the matching documents is infeasible. Using blocking techniques [11, 7] it is possible to avoid comparisons between documents that are highly unlikely to belong to the same entity. To avoid unnecessary comparisons we exploit textual similarity as well as inter-document references to form blocks/clusters of documents that are highly likely to belong to the same entity. Once such clusters are formed, we resolve the entities in each cluster using R-Swoosh based Iterative Match-Merge [8]. Finally we consolidate the results from each cluster using graph-clustering, i.e., by finding connected components in a graph of documents with edges defined using the partial-entity assignments (see Section 4.4).

3.2 Motivating Example

Consider the 11 documents in the collection $D = \{d_1, d_2, \dots, d_{11}\}$ containing information about 4 entities, as depicted in Figure 1 (and later used in Figures 2, 4, and 5). The documents d_1 to d_4 belong to entity e_1 , d_5 to d_7 belong to e_2 , d_8 to d_{10} belong to e_3 , and d_{11} belongs to e_4 . Given D and a *Match* function, the goal is to determine $E = \{e_1, e_2, e_3, e_4\}$ such that $e_1 = \text{Merge}(d_1, d_2, d_3, d_4)$, $e_2 = \text{Merge}(d_5, d_6, d_7)$, $e_3 = \text{Merge}(d_8, d_9, d_{10})$, and $e_4 = d_{11}$. The documents are of 4 types obtained from 4 different sources: Income Tax documents (PAN¹), Voter-Card (VOT), Driving-Licence (DL), and Bank Account Details (BAN). The attributes of the documents are *Document ID*, *Name*, *Email ID*, *Phone Number*, *Date of Birth*, *Address*, *Proof ID*, and *Document Details*. Documents d_1 and d_3 make explicit reference to d_2 through the *Proof ID* attribute. Documents d_7 and d_{11} make implicit reference to d_6 and d_9 respectively through the *Document Details* field. Documents d_3 and d_4 have high textual similarity, i.e., the values of their corresponding attributes are very similar. Same is the case with documents d_5 and d_6 , documents d_8 and d_9 , and documents d_8 and d_{10} .

Note that we have assumed that the schemas of different document types have already been matched, i.e., we already know which fields in each document refer to a person’s Name. When documents are completely unprocessed, even this can be a difficult problem; however, techniques for schema mapping are already available, such as [12].

The Match function is based on a disjunction of rules shown in Figure 2. A rule is satisfied when the Boolean functions based on the tick-marked columns are all *true*. For example, for two documents r and r' , Rule 2 reads: *sameName*(r, r') AND *sameDateOfBirth*(r, r') AND *sameAddress*(r, r'). Here, *sameName*(r, r'), *sameDateOfBirth*(r, r') and *sameAddress*(r, r') are Boolean functions. Two documents r and r' match based on Rule 2 when this clause is satisfied, and two documents “match” if at least one of the five rules is satisfied.

Note that each of the Boolean functions such as *sameName*() may be implemented by checking for exact textual match or a approximate match using some similarity measure. Now we briefly describe the major steps our **ERLD** algorithm using the above motivating example.

¹Income-tax ‘permanent account number’ in the Indian context

3.3 Major steps of ERLD

1. **DT: Document Traversal to exploit links between documents:** If a document makes a reference to some other document, then the two documents may belong to the same real-world entity. For example, a person may use her *driving license number* as a proof of her name/identity to get her passport, or a blog profile of a person may link to her twitter account. Such references help to identify some of the documents belonging to the same entity which in turn help to resolve it. This is particularly helpful when two documents belonging to the same entity do not have sufficient attributes-based similarity.

For example, a person may use her maiden name in her driving license, which in turn is used as identity proof (together with a marriage certificate) while applying for a passport. The driving license and passport may not match purely based on attribute similarity, but the inter-document reference forces them to be assigned to the same entity.

During Document Traversal we traverse the links between documents as defined by *referential attributes*. For example, documents d_1 , d_2 , and d_3 in Figure 2 are linked to each other. Such links can be captured through DT, and based on rules Rule 1 and Rule 3, which explicitly include a check for two documents being related by references (the *IsInTraversalSet()* function, which is true if two documents are connected by a path of references), it can be established that the documents belong to the same entity. We discuss Document Traversal in detail in Section 4.1.

2. **LSH: Similarity-based hashing to exploit textual similarity between documents:** Documents with similar content are likely to belong to the same real-world entity. For example, if the values of *name*, *address*, and *phone number* attributes are same in two documents (as is the case with documents d_3 and d_4 in Figure 2), it is very likely that the documents belong to the same real-world entity. On the other hand, if only *name* matches, and *address* and *date-of-birth* differ, the two documents are less likely to belong to the same entity. To avoid unnecessary comparisons between documents with very low textual similarity, we use *MinHash based Locality Sensitive Hashing* [13, 14] to cluster similar documents, and perform comparisons only amongst documents belonging to the same cluster. As a result, after DT and LSH, documents that are not textually similar and are not linked to each other through references are unlikely to be even compared. We describe how clusters are computed using LSH in Section 4.2 .
3. **IMM: Iterative Match-Merge to handle incomplete information in documents:** A document obtained by merging two matching documents may contain more information about the entity than the information revealed by either of the two documents when they are considered separately. To exploit this, we use *Iterative Match Merge* (refer to Section 4.3) based on the R-Swoosh algorithm described in [8]. For the example in Figure 2, documents d_5 and d_6 do not satisfy any of the rules (they only satisfy *sameName*(d_5, d_6) and *sameAddress*(d_5, d_6)), and hence don't match. Similarly d_5 and d_7 don't match (they satisfy *sameName*(d_5, d_6), *samePhoneNumber*(d_5, d_6), and *sameDateOfBirth*(d_5, d_6)). But d_6 and d_7 (linked through DT) match based on Rule 1. When these two matching documents are merged to get a new document (say, d_{67}), we now know the values of five attributes (*name*, *email-id*, *phone number*, *date-of-birth*, and *address*) for d_{67} . We have additional information due to the values of *phone number* and *date-of-birth* for an entity coming from document d_7 , and two different addresses for the same entity coming from d_6 and d_7 . This additional information can be useful in future matching operations. The merged document d_{67} has sufficient information to match with d_5 based on Rule 2. (Note that d_5 and d_{67} are considered for comparison because they are in same block due to the combined effect of LSH and DT. In other words, LSH puts d_5 and d_6 in same block while DT links d_6 to d_7 . So all the three documents end up being in the same block. The details of this process are described in Section 4.)
4. **CC: Connected Components** The combination of DT and LSH results in blocks of documents that are resolved into entities. Still, the possibility remains for documents belonging to a single entity to get mapped to more than one block. For example, one LSH block due to *Name*, and another due to *Address*. Yet the two 'partial entities' formed in each such block are still connected by the fact that they both contain overlapping documents. Therefore, we finally consolidate *partial entities* emerging from all the blocks by computing connected components in a graph of documents where an edge exists between two documents if they belong to the same partial entity. If a pair of partial entities e_a and e_b happen to share at least one of the original documents d_i , they end up being in the same connected component and all the documents in them get merged.

Summary: In a scenario where a link between two documents means that both the documents belong to the same entity, and all documents belonging to the same entity are also linked to each other, then the entity can be resolved only on the basis of DT. However, when the documents belonging to the entity form more than one component in the graph of document-linkages, we need other ways to connect all the documents belonging to the same entity. For such cases, a blocking-scheme based on a combination of DT and LSH can help, where DT helps to discover links based on referential attributes and LSH helps to discover potentially matching documents based on non-referential attributes. For the example in Figure 2, d_3 and d_4 get hashed to the same block on the basis of LSH. Documents d_1 and d_2 do not have high textual similarity with either d_3 or d_4 , and hence using only LSH for blocking, no single block would form containing d_1 , d_2 , d_3 , and d_4 , and the entity e_1 would not get completely resolved. It is because of DT that d_1 and d_2 get linked to d_3 . The blocking scheme based on combination of DT and LSH leads to at least one such block where all the four documents belonging to e_1 co-exist, and then using IMM on all the four documents results in the desired entity e_1 .

We consider two different scenarios in which Entity Resolution is encountered in practice: 1) *Batch mode*, when an entire collection of documents need to be resolved within themselves, and 2) *Incremental mode*, when a new, relatively small collection of documents need to be added to an already resolved entity-document collection. We first describe in detail the batch mode process (ERLD) and then describe how the batch mode solution can be modified to perform Incremental Entity Resolution (Incremental ERLD).

4 ERLD Algorithm Details

We now describe the steps of the ERLD algorithm (see Figure 3 and Algorithm 1) in the order they are performed.

Algorithm 1: ERLD Algorithm

Input: $D = \{d_1, d_2, \dots, d_N\}$, raw documents.
Parameters: $m = \#Hash\ functions\ required\ to\ get\ a\ bucket\ id$; $n = \#BucketIds\ generated\ per\ document$.
Output: $E = \{e_1, e_2, \dots, e_K\}$, resolved entities.

```

// key is bucket-id, value is list of documents;
1 buckets  $\leftarrow$  empty hash table;
2 bucketIds  $\leftarrow$  empty array;
3 Build inverted index;
4 for  $i \leftarrow 1$  to  $N$  do
5    $d_i.traversalSet \leftarrow getTraversalSet(d_i)$ ;
6    $d_i.minhashSignature \leftarrow getMinhashSignature(d_i)$ ;
7    $bucketIds \leftarrow getBucketIds(d_i.minhashSignature)$ ;
8   for  $j \leftarrow 1$  to  $size(bucketIds)$  do
9     bucket  $\leftarrow$  empty list;
10    bucket  $\leftarrow buckets.get(bucketIds[j])$ ;
11    bucket.add( $d_i$ );
12    bucket.add( $d_i.traversalSet$ );
13    buckets.put( $bucketIds[j]$ , bucket);
14  end
15 end
16 docs  $\leftarrow$  empty list;
17 partialEnts  $\leftarrow$  empty list;
18 foreach key in buckets.keySet do
19   docs  $\leftarrow buckets.get(key)$ ;
20   partialEnts  $\leftarrow RSwoosh(docs)$ ;
21   edges  $\leftarrow generateEdgeList(partialEnts)$ ;
22   edgeList.add(edges);
23 end
// key is component-id, value is list of documents
24 conComps  $\leftarrow$  empty hash table;
25 conComps  $\leftarrow ConnectedComponents(edgeList)$ ;
26 int  $i \leftarrow 1$ ;
27 foreach component in conComps do
28   docs  $\leftarrow component.getDocs()$ ;
29    $e_i \leftarrow merge(docs)$ ;
30    $i \leftarrow i+1$ ;
31 end

```

4.1 Document Traversal

Documents are indexed by a primary-key-based database, where a concatenation of the document type and a unique attribute for that document type is used as primary key. For example, PAN11 indicating document with unique PAN number 11 in the set of PAN

Bucket-ID	Documents grouped by LSH (with their traversal set) Document-ID: {Traversal Set}	Resulting Documents in Bucket	Partial Entities	Edge List	Connected Components	Final Resolved Entities
b ₁	d ₃ :{d ₂ ,d ₁ }, d ₄ :{}	d ₁ , d ₂ , d ₃ , d ₄	e ₁ ': Merge(d ₁ ,d ₂ ,d ₃ ,d ₄)	d ₁ -d ₂	c ₁ : {d ₁ ,d ₂ ,d ₃ ,d ₄ }	e ₁ : Merge({d ₁ ,d ₂ ,d ₃ ,d ₄ })
b ₂	d ₅ :{d ₇ }, d ₆ :{d ₆ }	d ₅ , d ₆ , d ₇	e ₂ ': Merge(d ₅ ,d ₆ ,d ₇)	d ₁ -d ₃	c ₂ : {d ₅ ,d ₆ ,d ₇ }	e ₂ : Merge(d ₅ ,d ₆ ,d ₇)
b ₃	d ₈ :{d ₉ }, d ₉ :{d ₁₁ }	d ₈ , d ₉ , d ₁₁	e ₃ ':Merge(d ₈ ,d ₉) e ₄ ':d ₁₁	d ₁ -d ₄ d ₅ -d ₆ d ₅ -d ₇	c ₃ :{d ₈ ,d ₉ ,d ₁₀ } c ₄ :{d ₁₁ }	e ₃ : Merge(d ₈ ,d ₉ ,d ₁₀) e ₄ :d ₁₁
b ₄	d ₈ :{d ₉ }, d ₁₀ :{}	d ₈ , d ₁₀	e ₃ '':Merge(d ₈ ,d ₁₀)	d ₈ -d ₉ d ₈ -d ₁₀ d ₁₁ -d ₁₁		

Figure 4: Example of ERLD Flow after DT and LSH have been done for documents in Figure 1

documents. Additionally, an inverted index is built over all the *referential attributes* of the documents so that these attributes become searchable.

Consider the documents as nodes in a graph and the references between documents as directed edges in the graph. The direction of an edge is determined as follows: if document d_i refers to document d_j either explicitly or implicitly, then the edge is directed from d_i (the source document) to d_j (the referred document). Going downwards in the graph (starting from a node) means traversing the graph along the direction of the edges, i.e., from the source documents towards the referred documents, whereas, going upwards means traversing in the direction opposite to the edge-direction, i.e., from the referred document to the source document.

We maintain a *traversal set* for a document, which is a set of documents connected to it via some path or paths. The *traversal-set* for a document is generated using a combination of Downstream Traversal (DST) and Upstream Traversal (UST), which we describe next:

Downstream Traversal (DST): Starting from a document, DST captures all the documents reachable from it going in the downward direction, considering only the edges due to explicit references. The edges due to the explicit references made by a document are captured by searching the values of the *explicit referential attributes* in the document via the primary key of the database containing all the documents.

All the documents obtained by DST on a given document form the *downstream traversal set* for it. For the example in Figure 2, considering only the explicit references, the downstream traversal set for d_1 is $\{d_2\}$, and for d_3 is $\{d_2\}$. The downstream traversal set for rest of the documents is \emptyset . As another example, considering only the explicit references, suppose r_1 refers to r_2 , r_2 refers to r_3 , and r_3 refers to r_4 and r_5 . Then, the downstream traversal set for r_1 is $\{r_2, r_3, r_4, r_5\}$. The downstream traversal set for r_2 is $\{r_3, r_4, r_5\}$, and for r_3 is $\{r_4, r_5\}$. The downstream traversal set for r_4 and r_5 is \emptyset .

Note however, in DST we do not capture the documents that are implicitly referred in a document because we do not know beforehand which part of the value of the implicit referential attribute should be searched for using the primary key of the document database. For the example in Figure 1, for d_6 , the value of the implicit referential attribute *Document Details* is “Driving License ID:DL77”. Here, we do not know beforehand that “DL77” is the key which should be searched for. Implicit references are captured in a different manner through UST, exploiting our inverted index, which we discuss next.

Upstream Traversal (UST): Starting from a document, UST captures all the documents reachable from it going in the upward direction, considering the edges due to both explicit and implicit references. Apart from the direction of traversal, UST is different from DST because it also captures the implicit references. For doing UST, an inverted index over the explicit and implicit referential attributes of all the documents is required. In UST, the primary key and the values of the explicit referential attributes for the given starting document are searched using the inverted index. The documents retrieved through this search are put in the *upstream traversal set* for the document.

For example, consider a situation where r_1 makes an implicit reference to r_2 , which in turn makes implicit references to r_3 and r_4 . Also, assume that r_1 makes an explicit reference to r_5 . Then the upstream traversal set after a single step of UST for r_2 will be $\{r_1\}$, and for both r_3 and r_4 , the upstream traversal set will be $\{r_2\}$. The upstream traversal set for r_5 will be $\{r_1\}$, and the upstream traversal set for r_1 will be \emptyset . For the example in Figure 2, the upstream traversal set for d_7 is $\{d_6\}$, for d_9 is $\{d_{11}\}$, for d_2 is $\{d_1, d_3\}$. The upstream traversal set for the rest of the documents is \emptyset .

For each document, DST and UST are used to get a traversal set (see line 5 in Algorithm 1): First, a single step of DST is done for the starting document, and the downstream traversal set thus obtained is added to the traversal set. This is followed by a single step of UST for all the documents in the traversal set and the starting document itself. The documents retrieved from this UST step are used to augment the traversal set. This process of single step of DST followed by a single step of UST is one *DST-UST step*. This DST-UST step can be repeated a number of times on the documents that get added after each DST-UST step to the *traversal set*, with a constraint on the

maximum number of DST-UST steps and/or the maximum number of documents retrieved after a single step of UST for a document.

If the number of documents retrieved in a single step of UST for a document is more than a threshold, then the retrieved documents are not added to the traversal set. This constraint is particularly important in some cases. For example, a *homepage-url* of a company may appear as an implicit reference in documents belonging to many of its employees. If this *url* also appears as a value of some explicit referential attribute in some document in the database, then UST for this document will retrieve a large number of documents. Most of these documents are very unlikely to belong to the same entity to which the starting document belongs, and should be discarded as noisy results. The constraints can sometimes come from the domain knowledge. For example, if it is known that there can be a maximum of 5 types of documents for an entity and one document of the entity can only refer to another document for the same entity, then the maximum DST-UST steps required to capture all the documents connected to any of the documents is 4. (This happens when the graph is such that all the 5 nodes are connected to each other as $r_1-r_2, r_2-r_3, r_3-r_4, r_4-r_5$, i.e. r_1 refers to r_2 , r_2 refers to r_3 , and so on.)

After traversal, any document has a traversal set associated with it, which is a set of documents potentially belonging to the same entity as the document. To get further potential matches for any document and avoid unnecessary comparisons, we use Locality Sensitive Hashing (LSH).

4.2 Locality Sensitive Hashing

Consider a set of words created from the values of the attributes (except the referential attributes) in a document. If two documents have a large number of words in common, i.e., the Jaccard similarity coefficient of the corresponding sets of words for two documents is high, they should be considered for comparison as compared to two documents which do not have many words in common.

LSH hashes the documents to *buckets* such that the probability that two documents get hashed to the same bucket is equal to the Jaccard similarity coefficient of the set of words corresponding to the two documents [15]. The buckets are formed such that documents with high Jaccard similarity are more likely to end up in the same bucket. Note: The traversal set of a document is also put in the buckets the document belongs to, so as to retain blocking information from the previous DT stage.

Each bucket is a key-value pair, where ‘key’ is the bucket-id, and value is the set of documents which get hashed to this ‘key’ along with their traversal sets. So, after hashing has been done for each document, each bucket contains documents which either have high textual similarity, or share explicit and/or implicit references (see lines 8 to 13 in Algorithm 1). The documents which belong to the same bucket are considered for Iterative Match-Merge, which we discuss in Section 4.3.

We use standard LSH via minhashing: Each word occurring in any of the documents is mapped to a unique integer. For each document, we consider the set of integers s corresponding to the set of words it contains. The minhash for the set s is calculated as the $\min((ax_i + b) \bmod p), \forall x_i \in s$ (where a and b are random integers, and p is a large prime). Different combinations of a and b determine different hash functions (see [13] for details). The probability that two documents, d_i and d_j (each containing sets of words w_i and w_j respectively) have the same minhash values for a minhash function is equal to the Jaccard similarity coefficient, $J(w_i, w_j)$.

For each document, we generate $m \times n$ hash values using $m \times n$ minhash functions. From these $m \times n$ hash values, m of the hash values are combined together using string concatenation to get a single *bucket-id*, resulting in a total of n such bucket-ids for each document. Due to this concatenation to get bucket-ids, the probability that two documents, d_i and d_j , have at least one same bucket-id is equal to $1 - (1 - (J(w_i, w_j))^m)^n$.

These two steps, i.e., concatenation and ‘or-ing’ of hash functions, reduce the probability that two documents, d_i and d_j , with very low Jaccard similarity (close to 0) will get hashed to the same bucket, compared to the earlier probability of $J(w_i, w_j)$. Similarly, the probability that two documents d_i and d_j with very high Jaccard similarity (close to 1) will get hashed to the same bucket is increased.

For the example in Figure 2, documents d_5 and d_6 have high textual similarity and end up being in at least one same bucket (bucket b_2 in Figure 4). Also, since the traversal set for d_7 is $\{d_6\}$, the document d_6 also comes in the same bucket along with d_7 . So, the blocking based on DT followed by LSH helps in resolving the entity e_2 (refer Figure 4 and 5).

The documents in a bucket (cluster) go through the Iterative Match-Merge (IMM) process using R-Swoosh [8], which we discuss next.

4.3 Iterative Match-Merge

We use R-Swoosh, an iterative match-merge process to resolve the documents in each bucket, using the given *Match* function and the *Merge* procedure (see line 20 in Algorithm 1). We briefly introduce R-Swoosh as described in [8], and implemented in SERF². Consider two sets of documents, I and I' . To begin with, set I contains all the documents from a bucket, and set I' is empty. I' contains the documents that have all been compared with each other. R-Swoosh iterates over the documents in set I . Iteratively, a document d is removed from I , and compared to every document in I' . If no matching document is found for d in I' , it is added to I' . On the other hand, if a document d' is found in I' , such that d matches d' , then d' is removed from I' , and a new document obtained through *Merge*(d, d') is added to I . Although, d' does not match any other document in I' , the new merged document may match some document in I' . The new

²<http://infolab.stanford.edu/serf/#soft>

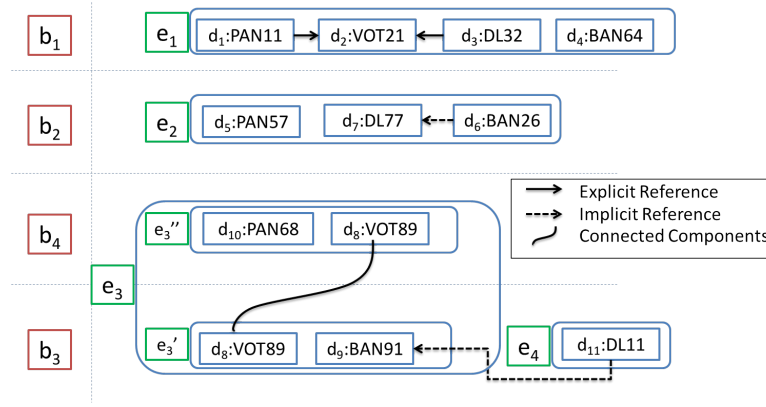


Figure 5: Sample buckets for the example in Figure 1

document is added to I , so that it is compared with every other document either in I or I' . In the end, the set I is empty, and I' contains the final result of R-Swoosh on the documents in the bucket.

It is to be noted that we do not compare a pair of documents twice, and rather maintain two sets: one of *matching pairs* and another of *non-matching pairs*. If a pair of documents (which has already been compared in a bucket) is encountered again (in another bucket), the Match function is not computed again. If a pair of documents exists in the set of matching pairs, the value of Match function is taken to be `true`; if it exists in the set of non-matching pairs, the value is taken to be `false`; else the Match function is computed for that pair, and based on the value of the Match function, either the set of matching pairs or that of non-matching pairs is updated.

The documents for one real-world entity can co-occur in multiple buckets. So, same entity can be obtained (as a result of **IMM** process) from multiple buckets. Also, due to the probabilistic nature of LSH, there may not exist a bucket, which has *all* the documents belonging to an entity. On the other hand, a bucket can have more than one entity (The number of entities in a bucket is equal to the final number of documents in the set I'). We call the entities obtained from each bucket as *partial entities*. So, it is required to combine the results (partial entities) from all the buckets to get the final resolved view of entities, as discussed below.

4.4 Connected Components

The results from the buckets need to be combined so that if any two partial entities belonging to different buckets have a common document, then the entities are combined to get one entity by applying the *Merge* operation on the two partial entities (which is equivalent to merging all the documents belonging to the two partial entities). The **IMM** process ensures that a document cannot belong to more than one partial entity within the same bucket (because, as soon as a document matches some other document, a new document is created by merging the original ones and the original ones are removed). Hence, this merging is required for partial entities sharing a common document but coming from different buckets, rather than for partial entities within the same bucket.

For the example in Figure 2, documents d_8 , d_9 , and d_{10} belonging to the same entity e_3 do not share any references to each other. As shown in Figures 4 and 5, documents d_8 and d_9 (which has document d_{11} belonging to a different entity e_4 linked to it through UST) end up in bucket b_3 based on textual similarity. Similarly, documents d_8 and d_{10} end up being in the bucket b_4 . So, there is no such bucket which has all the documents belonging to e_3 in it. In bucket b_3 , IMM helps to produce a partial entity $e_3' = \text{Merge}(d_8, d_9)$. In bucket b_4 , IMM helps to produce another partial entity $e_3'' = \text{Merge}(d_8, d_{10})$. Now these two partial entities need to be somehow merged:

We formulate the problem of combining the partial entities from different buckets into the problem of finding the *connected components (CC)* in an undirected graph. In a distributed setting, connected components has been used to assign entity ids to matching documents in a similar manner in [16].

Consider the documents as *nodes* of an *undirected graph*. There is one node in the graph for each document in the collection D . We create the *edges* in the graph as follows: For each partial entity, randomly select one of the nodes from it as a *central node*. Then, add an edge between the central node and each of the remaining nodes of the partial entity. This ensures that all the nodes in any partial entity are *connected* to each other (through the central node). Consider bucket b_1 in Figure 4, which has a partial entity $e_1' = \text{Merge}(d_1, d_2, d_3, d_4)$. Selecting d_1 as the central node for e_1' , the edge-list from e_1' will be $\{d_1-d_2, d_1-d_3, d_1-d_4\}$.

The way in which the graph is constructed ensures that all the nodes in a partial entity are connected. Also, if any two partial entities have a node (document) in common, then all the nodes belonging to the two partial entities are connected, and belong to the same entity. So, one connected component in the graph corresponds to one entity. Hence, by finding the connected components in the graph constructed as described above, we can combine the results of the buckets. More precisely, the results from different buckets can be combined to get the resolved entities (see lines 21 to 29 in Algorithm 1) in the following way: (1) Find the connected components in a graph constructed from the partial entities from all the buckets. (2) Get the final resolved entities by applying the *Merge* function on the

nodes (documents) in each connected component.

For the example in Figure 5, the pair d_8 - d_9 gets added to the edge-list from bucket b_3 and the pair d_8 - d_{10} gets added to the edge-list from bucket b_4 . When CC is applied on the edge-list, d_8 - d_9 - d_{10} emerges as a single connected component c_3 . Then, the Merge function is applied on d_8 , d_9 , and d_{10} to get the final resolved entity $e_3 = \text{Merge}(d_8, d_9, d_{10})$.

Note that the above procedure is functionally equivalent to computing connected components the smaller graph of partial entities, where an edge denotes two entities sharing some document. However, computing these edges itself requires traversing and sorting the documents and is therefore not computationally more efficient than the above approach.

5 Incremental ERLD

We now show how the ERLD algorithm above can be modified to operate incrementally. In other words, we describe how a previously resolved entity-document collection can be resolved with a new set of documents *without* having to re-resolve the entire collection.

First, the inverted-index over the entire document collection is updated to include the new set of documents. Next, Document Traversal (DT) is performed for each new document. A single step of DST captures documents reachable ‘downwards’ from each new document, from amongst the new set as well as the old collection. Similarly, a single step of UST for can discover references both from the new and old documents.

Now any old documents obtained through DST or UST are replaced by the corresponding previously resolved entities. Thus the traversal-set of a new document can now contain both documents as well as entities: Any documents in this set are necessarily from the new set of documents, and any entities are those that have been previously resolved.

DT is followed by LSH on the new set of documents. The bucket-ids created by the LSH process on the new set of documents may contain bucket-ids that were also created earlier LSH was earlier applied on the old documents. The ids of old documents that got hashed to such bucket-ids are retrieved from the previously created LSH-Index. The corresponding old resolved entities for these document-ids are retrieved. As a result we now have two types of buckets in the new LSH-index: those that include old entities (obtained either through traversal or LSH on a new document) and the remaining that contain only new documents. The former may lead to updating retrieved old entities in case some new documents need to be merged into them.

IMM is applied on the new documents as well any entities in each bucket. As in batch mode, partial entities are generated to be combined using CC. A partial entity in case of Incremental ERLD can be of one of the 3 types: (1) consisting only of new document(s), (2) consisting of new document(s) and old entity/entities, (3) a not updated old entity. If a partial entity is of first or third type, the edge-list is created in the same way as done in ERLD (Section 4.4). On the other hand, if a partial entity is of type-2, the set of document-ids consists of document-ids for the new documents and the document-ids contained in the old-entities. Edge-list is created for this set of document-ids by assuming one document (node) as a central node and the remaining nodes are connected to it. It is to be noted that, the CC step in Incremental ERLD, happens only on documents (nodes) which are part of the partial entities obtained during Incremental ERLD, and not all the documents (from the old batch and the new set). So, CC happens on the graph corresponding to all the new documents and possibly a few of the old documents. These partial entities consist of old entities and new documents. The edge-list for the old entities is created in the same way as it is done for partial entities in the batch-mode entity resolution as described in 4. The CC step may result in updating the entity-ids (and, in turn, the document-entity map) for some old documents; for example, previously separate entities may get merged because of fresh information obtained from newly arriving documents. The final resolved entities are obtained by applying the Merge procedure on the nodes (documents) in each connected component.

Next the LSH-index is updated by adding the new buckets (buckets with ids not present in the existing LSH-index) as well as adding new documents to previously existing buckets (documents that get mapped to previously existing bucket-ids). The document-entity map is also updated, which involves updating the entity-ids of some old documents, as well as adding document-ids and entity-ids for the new documents; in addition, certain previously resolve entities may disappear by getting merged with others. The updated LSH-index, document-entity map, and inverted index are re-used for the subsequent incremental ERLD steps, i.e., when fresh sets of documents arrive.

For the example in Figure 2, assume that document d_3 was not present when the first batch of documents was resolved. In the absence of d_3 , it is not possible to get d_1 , d_2 , and d_4 in the same bucket. As a result there would be two entities present corresponding to the entity e_1 in the previously resolved entity-document collection. One would be $e'_1 = \text{Merge}(d_1, d_2)$ (on the basis of traversal), and the other would be $e''_1 = d_4$. When d_3 arrives as a part of a new incremental set of documents, it will get linked to d_1 and d_2 (which map to the entity e'_1) on the basis of DT. Also, since d_3 has high textual similarity with d_4 , LSH on it will produce at least one bucket-id which has d_4 (the entity e''_1) in it. In this way, d_3 with its *traversal set* $\{e'_1\}$ will be present in a bucket with entity e'_1 . Hence, all documents and previously resolved entities belonging to the desired entity e_1 land up in the same bucket, so the IMM process in this bucket is able to successfully resolve the entity e_1 . At the same time, we have avoided re-resolving the entire collection, i.e., the previously resolved entities e_2 and e_3 are not even accessed.

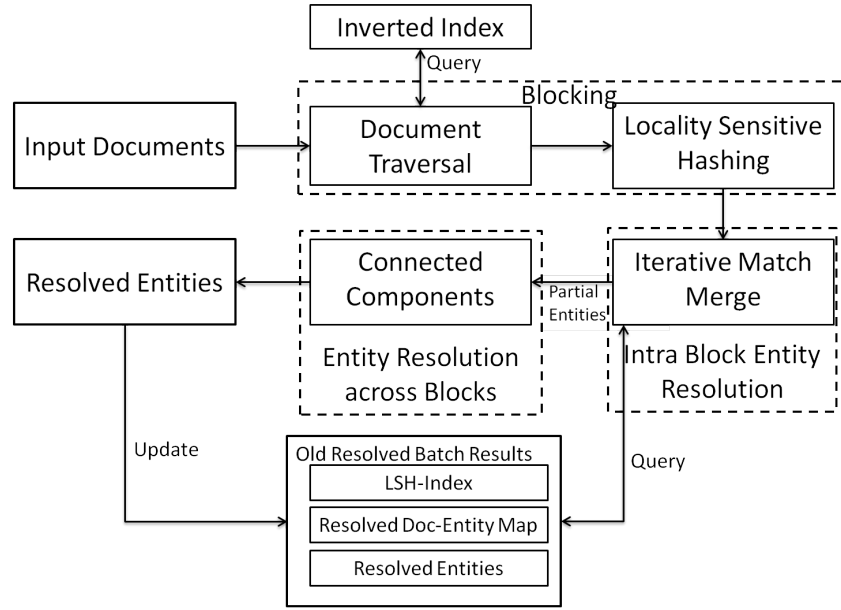


Figure 6: Incremental ERLD

6 Experimental Evaluation

We present performance (execution-time) and quality (precision, recall, and F1-score) results on two data-sets: a real-world database of companies and a large synthetically generated ‘residents’ database. We first describe the two data-sets and then present the benchmarks to prove the benefit of i) traversal through enhanced recall, ii) blocking (based on traversal and LSH) that enables scalability, and iii) incremental resolution that saves execution-time while maintaining quality as compared to the batch-mode.

Our implementation was in Java 1.6, and the experiments performed on a Linux (Ubuntu) server with Intel Xeon E7540 2GHz processor, 64GB RAM and 4 CPUs with 6 Cores each.

6.1 Synthetic Residents’ Data

We automated the generation of a synthetic ‘residents’ database of documents in the following manner: We began with 100,000 seed documents where a seed document has all the information about an entity. For each such entity, a maximum of 5 documents are created. These documents belong to one of the following 5 domains: Voter-Card, PAN Card, Driving-Licence, Bank Account Details, and Phone Connection Application. An entity can have a maximum of 1 document per domain. We control the creation of a document of a particular type for an entity using a random-number generator. To create a document from the seed document, values of some of the attributes of the person are retained. The values of the attributes of an entity across documents are varied by using different methods. For example, the address of the different documents for the same entity need not be same. Even if the address is same, a slight variation is inserted by skipping a few characters to introduce typographical errors. The first name of person is varied in some cases by omitting a few characters, swapping the first and last name, or omitting the middle name. Also, some of the attributes are not given any value. For example, e-mail id of a person may be present in 2 of his documents and absent in others.

Once the documents for an entity are created, documents are also created for some related entities, such as parent, child, spouse, neighbour, etc. This also adds ambiguity to the resolution of entities as documents of related entities have a considerably high textual similarity. Once all the documents for a person are generated, references are added between them in a random manner. The documents belonging to the same person share explicit references between them.

References between documents belonging to different entities are not present in this data-set. Note that such references would need to be treated differently from references between documents belonging to the same entity. In most cases document structure would make it easy to distinguish *explicit* references that point to documents of another entity, e.g., a field labeled ‘father’s passport number’. Sometimes however, and certainly for *implicit* references (i.e., in textual description fields), such a distinction may not be possible. We do not address (a) disambiguation of such references, nor do we exploit (b) the potential of collective entity resolution by using relationships between entities to improve performance, as in [6].

We generated 580,363 documents corresponding to 209,501 entities. The average traversal set size for a document is 1.59 with a range from 0 to 4. Out of these, the traversal set size for 158,708 documents is 0, for 142,770 documents it is 1, for 124,873 documents it is 2, for 87,374 documents it is 3, and for the remaining 66,638 documents it is 4.

Algorithm	Precision	Recall	F1 Score
References not used	1.00	0.96	0.98
References used	1.00	0.98	0.99

Table 1: Benefit of Traversal in Residents' Data Batch-mode ERLD. Number of documents = 580,363

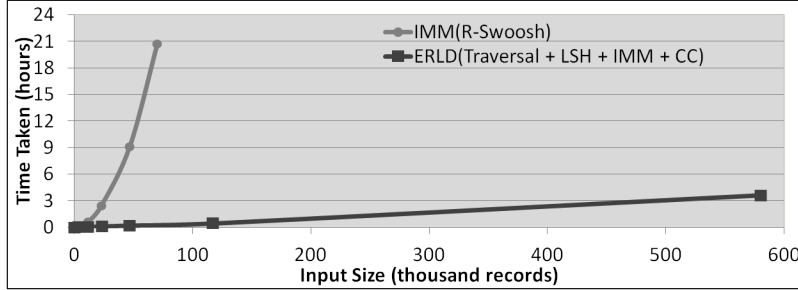


Figure 7: Scalability Analysis for Residents' Data

Benefit of Document References: Table 1 demonstrates the additional benefit of using inter-document references via DT during blocking of documents: Consider add the rule the rule *sameName(r,r')* AND *sameAddress(r,r')* AND (*isInTraversalSet(r,r')* OR *isInTraversalSet(r',r)*), which includes clauses testing for inter-document references. We witness a 2% increase in recall without any loss in precision when this rule exploiting inter-document references is used, as compared to the results without such a rule.

Performance gains from DT+LSH-based blocking: Blocking based on traversal and LSH has a much better execution-time performance (refer Figure 7) than IMM without any blocking. Whereas the Iterative Match-Merge (IMM) based approach takes about 21 hours to resolve 70,291 documents, our batch-mode entity resolution process takes less than 4 hours to resolve 580,363 documents without any change in the quality of results (both having precision = 1.00, recall = 0.95 for 70,291 documents).

Efficiency and accuracy of Incremental ERLD: The 580,363 documents were randomly distributed into 2 sets X and Y, such that X had roughly 95% (551,249) of the documents, and Y had the remaining 5% of the documents (29,114). From Table 2 it can be observed that first performing ERLD on a data-set(X) and then Incremental ERLD on the new batch of data(Y), results in saving of more than 2 hrs with respect to performing ERLD twice first on initial data-set (X) and then on combined data-set(X+Y), without any compromise in quality of results.

6.2 Companies' Data

This real-life data-set contained information about companies with attributes such as, Company Name, Email, Contact Phone, Company Fax, Address Line 1, Address Line 2, City, State, Postal Code, etc. There were a total of 669,245 documents; out of these 2196 pairs were manually labeled by domain experts, resulting in 1697 unique entities. A union of the 2196 pairs of documents gave us 3801 distinct documents (since some a documents were present in more that one pair).

For this data set we used a supervised SVM-based binary classifier (similar to [17]) as the *Match* function (instead of rules). for each pair of documents, features were computed as follows: Eight document attributes were used. The values of an attribute from two documents in a pair was compared using 7 string similarity metrics {Overlap Coefficient, Jaro-Winkler, SoftTfIdf, Soundex, Monge-Elkan, Jaccard Similarity, Cosine Similarity} each giving a score in the range of 0 to 1. This resulted in a 56(= 8x7) dimensional feature-vector for each pair of documents. We used 70% of the matching pairs (2196) as positive samples for the training set. All the document pairs other than those present in the labeled matching pairs ($^{3801}C_2 - 2196$) are assumed to be non-matching. Only, 3% of

	Precision	Recall	F1 Score	Time Taken (hh:mm)
ERLD(X+Y)	1.00	0.98	0.99	03:38
ERLD(X)	1.00	0.91	0.95	03:29
IncrementalERLD(Y)	1.00	0.98	0.99	01:20
ERLD(X)+ERLD(X+Y)	1.00	0.98	0.99	07:07
ERLD(X)+IncrementalERLD(Y)	1.00	0.98	0.99	04:49

Table 2: Incremental-mode ERLD for Residents' Data, documents resolved using Batch-mode:X=551,249, documents resolved using Incremental-mode:Y=29,114, Total number of documents (X+Y)=580,363

Algorithm	Precision	Recall	F1 Score
ERLD(X+Y)	0.974	0.878	0.924
ERLD(X)	0.973	0.800	0.878
IncrementalERLD(Y)	0.974	0.876	0.922

Table 3: Incremental-mode ERLD for Companies' Data. X=3605, Y=196, Total number of documents(X+Y)=3801

Algorithm	Precision	Recall	F1 Score
AllPairs+CC	0.971	0.885	0.926
LSH+IMM+CC	0.974	0.878	0.924

Table 4: Quality comparison on real-world Companies' Data: LSH+IMM+CC vs. AllPairs+CC. Number of documents = 3801

such pairs were taken as negative samples for training. An SVM with linear kernel function and cost factor $C = 0.06$ was used.

ERLD vs all-pairs comparison: Our ERLD algorithm based on blocking rather than exhaustive all-pairs comparison using a Match function, which in this case was our machine-learned classifier. We observe precision-recall and time-taken of two approaches, viz., (i) Compare each document with every other document. Create an edge for every matching pair and apply connected components on such a graph to combine all the documents which are part of same entity. (ii) ERLD, i.e., LSH followed by Iterative Match-Merge followed by Connected Components to resolve entities. (Note: there is no DT stage since this data set does not contain references).

Figure 8 shows that the time taken by the second approach is considerably less than the time taken by the first approach and Table 4 shows that this is achieved without any significant loss in the quality of the results.

Incremental vs Batch on Companies' data: Nevertheless, as depicted in Table 3, incrementally resolving the set Y (196 documents) against previously resolved set X (3605 documents) gives the same precision-recall as obtained by resolving all the documents (X+Y) together. Of course, the performance benefits of incremental vs batch resolution were less significant (3 minutes for Y vs 4 minutes for X+Y) than earlier, as the Companies' data set is considerably smaller than the synthetic residents data.

7 Related Work

Blocking for entity resolution has been often used [16]. Various iterative and non-iterative multiple blocking techniques for entity resolution have been discussed in detail in [7], such as Locality Sensitive Hashing, Q-gram Based Indexing etc. Locality Sensitive Hashing (LSH) based data-blocking for Iterative Record Linkage has been discussed in [18], which iteratively creates hash tables for records based on minhash functions and resolves the records being hashed to the same key. The resolved records at the end of each iteration are used as input records for next iteration. Various indexing (blocking) techniques for Entity Resolution like Sorted Neighborhood Indexing and Q-gram Based Indexing have been discussed in [11]. A set of blocking schemes that work for heterogeneous schema were proposed in [19]. In [20] "hints" are used after blocking, where the records that are more likely to get merged are compared before others, so that the partial results of Entity Resolution are available as they are generated. Our ERLD approach additionally includes inter-document references to enhance the benefits and quality of blocking.

We highlight the importance incremental-mode entity resolution where a small batch of documents is resolved, against an already

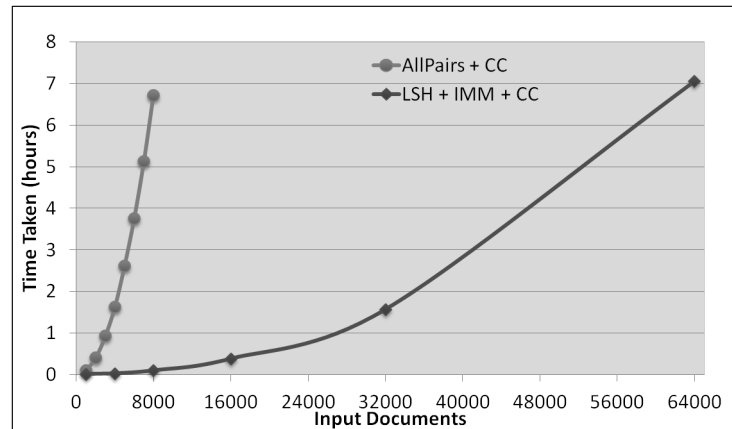


Figure 8: Scalability Analysis for Companies' Data

resolved entity collection, and propose an efficient solution. However, the concept of incremental processing is not new and has been proposed in [21]. In the domain of entity resolution, incremental duplicate detection has been attempted in [22]. However they process only one additional document at a time, while we process a batch of newly arrived documents and we perform complete entity resolution. Resolution of small set of entities has been attempted in [23, 24] for query interface and not in incremental manner, after bulk entity resolution has already taken place.

We have considered inter-document references that indicate that two documents belong to the same entity. However, such references could also indicate inter-entity relationships. In such cases our approach can possibly be used in as a precursor to collective entity resolution [6].

Our algorithm is based on traversal of the document graph. Graph traversal has been used in many other related domains, such as, ‘keyword search on graph data’ [25, 26]. Most of these approaches first convert the data into a graph, and then use various techniques for bi-directional graph traversal to discover a node that is reachable from the starting nodes. In our case, however, we never explicitly instantiate the entire document-graph; instead we only work on a small local neighbourhood of each document obtained using direct references (DT) and an inverted index (UST). Further, keyword-search algorithms include intermediate nodes in paths discovered between keyword nodes, resulting in a Steiner Tree problem. However, we assume that inter-document references are direct, rather than indirect via intermediate nodes.

8 Conclusions

We have shown how inter-document references can be leveraged to enhance the quality of entity resolution results through graph-traversal. We have then proposed a graph-traversal and Locality Sensitive Hashing based blocking scheme for a scalable solution to the Entity Resolution problem without any significant loss in the quality of results. We have further shown how to incrementally resolve a new set of documents against a pre-resolved entity-document collection without having to re-resolve all the documents.

References

- [1] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, “Duplicate record detection: A survey,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 19, no. 1, pp. 1–16, 2007.
- [2] M. Bilenko and R. J. Mooney, “Adaptive duplicate detection using learnable string similarity measures,” in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 39–48, ACM, 2003.
- [3] E. Rahm and H. H. Do, “Data cleaning: Problems and current approaches,” *IEEE Data Engineering Bulletin*, vol. 23, no. 4, pp. 3–13, 2000.
- [4] M. A. Hernández and S. J. Stolfo, “Real-world data is dirty: Data cleansing and the merge/purge problem,” *Data mining and knowledge discovery*, vol. 2, no. 1, pp. 9–37, 1998.
- [5] W. E. Winkler, “The state of record linkage and current research problems,” in *Statistical Research Division, US Census Bureau*, Citeseer, 1999.
- [6] I. Bhattacharya and L. Getoor, “Collective entity resolution in relational data,” *ACM Trans. Knowl. Discov. Data*, vol. 1, Mar. 2007.
- [7] D. Menestrina, *Matching and Unifying Records in a Distributed System*. PhD thesis, Stanford University, 2010.
- [8] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. Whang, and J. Widom, “Swoosh: a generic approach to entity resolution,” *The VLDB Journal*, vol. 18, no. 1, pp. 255–276, 2009.
- [9] C. J. Burges, “A tutorial on support vector machines for pattern recognition,” *Data mining and knowledge discovery*, vol. 2, no. 2, pp. 121–167, 1998.
- [10] S. Guo, X. L. Dong, D. Srivastava, and R. Zajac, “Record linkage with uniqueness constraints and erroneous values,” *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 417–428, 2010.
- [11] P. Christen, “A survey of indexing techniques for scalable record linkage and deduplication,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 24, no. 9, pp. 1537–1555, 2012.
- [12] J. Madhavan, P. A. Bernstein, and E. Rahm, “Generic schema matching with cupid,” in *VLDB Journal*, pp. 49–58, 2001.
- [13] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, “Min-wise independent permutations,” *Journal of Computer and System Sciences*, vol. 60, no. 3, pp. 630–659, 2000.

- [14] A. Andoni and P. Indyk, “Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions,” in *Foundations of Computer Science, 2006. FOCS '06. 47th Annual IEEE Symposium on*, pp. 459–468, 2006.
- [15] A. Rajaraman and J. Ullman, *Mining of Massive Datasets*. April 2010.
- [16] C. Sidl6, A. Garz6, A. Moln6r, and A. Bencz6r, “Infrastructures and bounds for distributed entity resolution,” in *9th International Workshop on Quality in Databases*, 2011.
- [17] W. W. Cohen, P. Ravikumar, and S. E. Fienberg, “A comparison of string distance metrics for name-matching tasks,” in *Proceedings of the IJCAI-2003 Workshop on*, (Acapulco, Mexico), pp. 73–78, August 2003.
- [18] H.-s. Kim and D. Lee, “Harra: Fast iterative hashed record linkage for large-scale data collections,” in *Proceedings of the 13th International Conference on Extending Database Technology, EDBT '10*, (New York, NY, USA), pp. 525–536, ACM, 2010.
- [19] G. Papadakis, E. Ioannou, C. Nieder6e, T. Palpanas, and W. Nejdl, “Beyond 100 million entities: large-scale blocking-based resolution for heterogeneous data,” in *Proceedings of the fifth ACM international conference on Web search and data mining*, pp. 53–62, ACM, 2012.
- [20] S. E. Whang, D. Marmaros, and H. Garcia-Molina, “Pay-as-you-go entity resolution,” *IEEE Trans. on Knowl. and Data Eng.*, vol. 25, pp. 1111–1124, May 2013.
- [21] P. Agarwal, G. Shroff, and P. Malhotra, “Approximate incremental big-data harmonization,” in *Proceedings of the 2nd IEEE International BigData conference*, IEEE, 2013.
- [22] M. Fisichella, F. Deng, and W. Nejdl, “Efficient incremental near duplicate detection based on locality sensitive hashing,” in *Database and Expert Systems Applications*, pp. 152–166, Springer, 2010.
- [23] P. Christen and R. Gayler, “Towards scalable real-time entity resolution using a similarity-aware inverted index approach,” in *Proceedings of the 7th Australasian Data Mining Conference-Volume 87*, pp. 51–60, Australian Computer Society, Inc., 2008.
- [24] I. Bhattacharya, L. Getoor, and L. Licamele, “Query-time entity resolution,” in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 529–534, ACM, 2006.
- [25] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, “Keyword searching and browsing in databases using banks,” in *Data Engineering, 2002. Proceedings. 18th International Conference on*, pp. 431–440, IEEE, 2002.
- [26] B. Kimelfeld and Y. Sagiv, “Finding and approximating top-k answers in keyword proximity search,” in *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 173–182, ACM, 2006.